ALMA MATER STUDIORUM
UNIVERSITA' DI BOLOGNA

CAMPUS DI CESENA
SCUOLA DI INGEGNERIA E ARCHITETTURA

CORSO DI LAUREA IN INGEGNERIA ELETTRONICA,
INFORMATICA E TELECOMUNICAZIONI

INTELLIGENZA ARTIFICIALE E GIOCHI:
L'ALGORITMO MINMAX

Elaborata nel corso di: Fondamenti di Informatica B

Tesi di Laurea di:
ANDRI VLADI

Relatore:
Prof. ANDREA ROLI

ANNO ACCADEMICO 2017-2018
SESSIONE III

Abstract

MINMAX è un algoritmo impiegato in diverse discipline, tra cui: teoria dei giochi, teoria delle decisioni e IA. Viene usato per la ricerca delle azioni migliori assumendo scelte ottimali da parte degli agenti coinvolti e sfrutta una struttura ad albero ed una funzione euristica per cercare una soluzione in un contesto di conflitto tra più parti. Costruisce l'albero fino a una profondità prefissata, identificata dal numero di azioni alternate che si sceglie come limite di ricerca e usa la funzione euristica per valutare gli stati all'ultimo stadio della ricerca, valori che propaga poi fino allo stato iniziale, confrontandoli con quelli degli altri stati, finché non ha valutato tutti gli stati adeguati disponibili, trovando cosi il percorso migliore. Il potenziamento principale a questo algoritmo è realizzato tramite la potatura Alfa-Beta che sfrutta due variabili chiamate appunto Alfa e Beta, che creano un finestra di valori minimi e massimi ammissibili oltre i quali le posizioni non vengono considerate, riducendo la dimensione dell'albero e permettendo così una ricerca più profonda.

Indice

Introduzione

I giochi sono uno dei passatempi preferiti di tante persone; alcuni però, richiedono più impegno rispetto ad altri. La ragione di questo, risiede nella necessità di pensare intensamente alle mosse durante lo svolgimento del gioco per poter avere una buona prestazione e raggiungere l'obbiettivo preposto, e in alcuni casi la presenza di uno o più avversari rende questo compito ancora più difficile. Il gioco degli scacchi per esempio, richiede l'applicazione di principi generali, di giudizio e di predizione da parte del giocatore in ogni momento.

Da molti anni ormai, studiosi in tutto il mondo lavorano per riuscire ad emulare l'intelligenza umana dando vita alla disciplina dell'Intelligenza Artificiale (IA).

Uno dei primi domini sfruttati dagli scenziati per la ricerca è proprio quello dei giochi, che nonostante la semplicità delle regole e degli obbiettivi, offre uno spazio complesso la cui navigazione sembra un'esclusiva dell'intelligenza umana. Si è pensato quindi che il riuscire a costruire una macchina in grado di giocare a scacchi, o a qualsiasi altro gioco altrettanto o più complesso, come o meglio di un essere umano, potesse avvicinare l'umanità alla creazione di un'IA in grado di risolvere qualsiasi tipo di problema le si presenti.

I metodi usati per il raggiungimento dell'arduo obbiettivo sono gli algoritmi, formalizzati dal matematico Alan Turing, considerato il padre dell'informatica e dell'IA.

L'algoritmo che vedremo in seguito si chiama MINMAX, che è stato scoperto in una disciplina della matematica applicata, chiamata teoria dei giochi. E' un metodo che si usa per minimizzare la massima perdita in situazioni di conflitto con soggetti rivali, in cui tutti gli agenti ricercano il guadagno massimo. Nel tempo sono stati introdotti altri algoritmi come Alfa-Beta, che vengono usati insieme a MINMAX per migliorare e velocizzare la ricerca di una soluzione. Verranno esposte brevemente anche alcune varianti di MINMAX come Negamax ed Expectiminmax, alcune ottimizzazioni di Alfa-Beta, e anche un algoritmo di più alto livello chiamato Monte-Carlo Tree Search (MCTS).

1 Tipologie di giochi

Il numero dei giochi esistenti è elevato ma si possono dividere in diverse categorie in base alle loro proprietà. Nel campo dell'IA vengono studiati principalmente giochi che hanno le caratteristiche degli scacchi: informazione perfetta, somma-zero, sequenziale, deterministico e non cooperativo [3].

1.1 Informazione perfetta e imperfetta

Nei giochi a informazione perfetta i giocatori sono a conoscenza di tutte le informazioni sullo stato presente e passato del gioco e sulle sue possibili continuazioni. Negli scacchi, che appertongono a questa categoria, le informazioni necessarie sono la posizione dei pezzi sulla scacchiera, le regole ufficiali e lo storico dello spostamento dei pezzi sulla scacchiera dall'inizio del gioco [1]. Con queste informazioni si possono fare tutte le deduzioni necessarie per giocare. Il poker invece è un gioco a informazione imperfetta poiché ogni giocatore, nel poker tradizionale, è a conoscenza solo delle proprie 5 carte che gli vengono date e non conosce quelle in mano ai propri avversari.

In entrambe le situazioni i giocatori usano delle strategie per decidere cosa fare. Una strategia definisce le azioni che verranno effettuate durante la partita per ogni situazione possibile in cui ci si possa trovare. I giochi a informazione perfetta usano una strategia pura dato che conoscono tutte le informazioni necessarie. Quelli a informazione imperfetta invece dovranno usare una strategia mista per un gioco ottimale, ovvero dovranno generare più strategie e dare ad ognuna di esse una valore di probabilità. La strategia pura può anche essere vista come facente parte di un insieme di strategia mista nel quale ha probabilità 1 e tutte le altre hanno probabilità 0 [4].

1.2 Somma zero e non somma zero

Nei giochi a somma zero a due persone se un giocatore vince significa che l'altro ha necessariamente perso. Se dovessimo usare la matematica per descrivere questo fenomeno, potremmo dare il valore 1 alla vittoria, 0 al pareggio e -1 alla sconfitta. Con queste premesse, la somma dei valori dell'esito della partita per i due giocatori deve sempre risultare 0. Quindi se per esempio negli scacchi il giocatore con i bianchi vince, viene valutato con 1 e poiché la somma deve essere 0, come abbiamo stabilito sopra, possiamo dedurre che l'altro giocatore ha

ottenuto una valutazione di -1 e quindi ha perso. Possiamo estendere il concetto anche a piu di due giocatori e dare una definizione in termini economici dicendo che i giocatori si pagano l'un l'altro e non c'è produzione o distruzione di beni [4]. Nel caso invece che che la somma dei valori dati alla fine della partita ai giocatori sia minore o maggiore di 0, ovvero diversa da 0, allora si parla di giochi a non-somma-zero. I giochi di intrattenimento sono generalmente del primo tipo.

1.3 Cooperativi e non cooperativi

La differenza tra giochi cooperativi e non cooperativi sta nella possibilità dei giocatori di poter effettuare accordi vincolanti sulle strategie da usare e/o la distribuzione della vincita, anche se questa opzione non fosse specificata nelle regole del gioco [6]. Quindi se ci si ritrova con un obbiettivo comune, due o più giocatori possono mettersi d'accordo e formare un'alleanza cooperando per raggiungere l'obbiettivo, questi sono i giochi cooperativi. I giochi non-cooperativi, che sono anche quelli più studiati, si fondano sull'indipendenza dei giocatori, che non comunicano e non collaborano tra di loro e perseguono in solitaria il proprio obbiettivo il cui raggiungimento va quasi sempre a discapito di tutti gli altri giocatori poiché ogni giocatore cerca di massimizzare il proprio guadagno, minimizzando di conseguenza quello degli altri. In queste circostanze, se il cambio di strategia di un giocatore (conoscendo le strategie degli altri e non potendole cambiare) non porta a un miglioramento della vincita, e questo è vero per tutti i giocatori, si è raggiunto l'equilibrio di Nash [7].

1.4 Deterministici e stocastici

I giochi deterministici possono essere risolti finitamente usando i valori dei parametri disponibili e le condizioni iniziali. Significa che siamo in grado, conoscendo lo stato presente e un'azione legale su tale stato, di determinare con certezza lo stato del gioco conseguente l'azione, e questo è vero per qualsiasi azione legale su ogni stato possibile dall'inizio alla fine del gioco. I giochi a informazione perfetta sono deterministici come gli Scacchi e Go.

I giochi stocastici possiedono caratteristiche di casualità che possono portare a risultati diversi date le stesse condizioni inziali e gli stessi valori dei parametri. In questa situazione non siamo in grado di prevedere l'esito di una nostra azione legale sulla posizione corrente durante un gioco, quindi è possibile che la stessa azione eseguita sulla stessa posizione in tempi diversi dia esiti diversi. I giochi a informazione imperfetta sono stocastici come Backgammon e Poker [3].

1.5 Sequenziale e simultaneo

Nei giochi con azioni simultanee non si sa che cosa farà l'altro giocatore finché non si è compiuta la propria scelta, non bisogna neccessariamente agire nello stesso momento, ma anche agire in tempi diversi non potendo sapere cosa farà l'altro è come se fosse simultaneo poichè la risoluzione del gioco avviene solo quando entrambi hanno effettuato la loro scelta e nessuno dei due può conoscere quella dell'altro prima di rilevare la propria. Questi tipi di giochi vengono rappresentati usando una matrice che contiene i valori relativi al risultato per ogni combinazione delle possibili azioni dei giocatori. Tipici giochi con azioni simultanee sono sasso-carta-forbici e il dilemma dei prigionieri [8]. A differenza dei giochi con azioni simultanee dove il valore del risultato è conosciuto per ogni singola azione, nei giochi sequenziali il valore del risultato finale dipende da una sequenza di decisioni [3]. Questo influisce sulle scelte poiché si è a conoscenza del comportamento precedente. Per trovare l'equilibrio di Nash nei giochi sequenziali, i giocatori devono identificare il giocatore con la strategia migliore e iterativamente dedurre la risposta migliore degli altri giocatori [9]. Tra i giochi sequenziali ci sono gli scacchi e il poker.

1.6 Complessità

“La proprietà complessità in relazione ai giochi, viene utilizzata per indicare due diverse misure, che chiamiamo la complessità dello spazio degli stati e la complessità dell'albero di gioco.” (Allis p.158)

Lo spazio degli stati è il numero delle posizioni legali che si possono raggiungere dallo stato iniziale mentre la complessità dell'albero di gioco è il numero di foglie nell'albero di ricerca delle posizioni iniziali [1]. In una partita di scacchi di 40 mosse ci sono 10^{120} numeri di foglie e 10^{43} posizioni legali [2].

2 Algoritmo MINMAX

Le prime tracce di MINMAX risalgono al 1912 e sono attribuite a Ernst Zermelo, il cui "articolo conteneva degli errori e non spiegava l'algoritmo in modo corretto, ma impostò le idee per l'analisi retrograda e propose quello che sarebbe diventato il teorema di Zermelo: ovvero che gli scacchi sono determinati." (Russell & Norvig p.190)

Ne illustriamo ora il funzionamento in modo informale, prendendo come esempio proprio il gioco degli scacchi.

Siccome gli esiti del gioco sono definiti (vittoria, sconfitta, pareggio) e le informazioni necessarie al raggiungimento del finale sono sempre disponibili (informazione perfetta), dovrebbe essere possibile scoprire la serie di mosse che porta ad un risultato preciso, considerando le proprie mosse e quelle dell'avversario fino alla fine del gioco, dove troviamo i diversi finali in cui tutti si trovano in uno dei tre stati possibili.

Tuttavia negli scacchi il numero di mosse e contromosse che si susseguono fino alla conclusione della partita è enorme come abbiamo visto nella sezione 2.6. E' chiaro quindi che un approccio del genere non sia praticabile. Ci dobbiamo affidare allora ad una funzione di valutazione approssimativa dello stato. Ogni stato in questo caso è una posizione dei pezzi sulla scacchiera e ogni movimento legale di un pezzo (mossa) modifica la posizione facendo avvenire una transizione da uno stato ad un altro. Usando una funzione di valutazione approssimativa potremmo attribuire un valore numerico ad ogni stato del gioco e cercare di raggiungere lo stato con il valore maggiore che identifica la vittoria.

Il problema è che essendoci due giocatori, ed assumendo che entrambi vogliano vincere, entrambi effettueranno la mossa che li avvantaggia di più, quindi mentre uno dei due cercherà gli stati che hanno la valutazione più alta l'altro farà l'opposto e sceglierà i valori minori. Quindi se per esempio prendessimo il punto di vista di un giocatore che chiamiamo A, alla prima mossa avremmo diverse scelte che portano a stati con varie valutazioni e tra queste dovremmo scegliere quella con il valore più alto. Ma in questo modo non consideriamo il fatto che dopo di noi sta all'altro giocatore, che chiameremo B, che avrà anch'esso un elevato numero di mosse disponibili e abbasserà la nostra valutazione il più possibile. In una situazione del genere la prima mossa del giocatore A deve tenere in considerazione la successiva del giocatore B e quindi dovrà scegliere una posizione in modo tale che quando tocca al giocatore B questi possa abbassare la valutazione il meno possibile.

Il giocatore A deve quindi cercare di minimizzare la massima diminuzione possibile della valutazione della posizione disponibile al giocatore B, poichè è quella la strada che il nostro avversario probabilmente percorrerà.

A questo punto abbiamo preso in considerazione solo le prime mosse per giocatore (livello di profondità 1 poichè si parte da 0) ma la partita è ben lontana dalla sua conclusione quindi non possiamo decidere la nostra mossa solo in base alla prima mossa del nostro avversario.

Si continua quindi con la stessa logica con le seconde mosse disponibili del giocatore A (livello di profondità 2) e poi del giocatore B (livello di profondità 3) e cosi via finchè non si raggiunge il livello di profondità desiderato.

Arrivati alla fine dobbiamo ora andare a ritroso per trovare la variante migliore che ci indicherà la prima mossa del giocatore A sfruttando i valori che abbiamo dato ad ogni

posizione usando la funzione di valutazione approssimativa.

Partendo dall'ultimo livello di profondità nel quale siamo arrivati, livello *n*, si sceglie lo stato con il valore maggiore quando ci troviamo in una posizione che è stata raggiunta grazie ad una scelta del giocatore A, dal punto di vista del quale stiamo eseguendo l'algoritmo,o lo stato con il valore minore nel caso in cui siamo arrivati a quella posizione in seguito a una mossa del giocatore B e lo portiamo al livello superiore che sarà *n*-1, significa che ignoriamo tutte le altre posizioni generate dalla stessa posizione di livello superiore che ha generato quella che abbiamo scelto e assegniamo il valore della posizione scelta alla posizione di livello superiore che l'ha generata. Si continua cosi finché non si raggiunge la posizione iniziale *n*-*n* (livello 0) che si vedrà assegnata il valore di una delle posizioni del livello più basso facendo apparire la variante principale poichè tutte le altre sono state progressivamente ignorate.

A quel punto possiamo effettivamente muovere il pezzo sulla scacchiera; poi sta al nostro avversario e si ripete tutto il procedimento di ricerca della variante migliore da capo.

Lo scopo è quindi non quello di scegliere necessariamente i valori maggiori negli stati intermedi, ma qualunque sequenza di stati che, considerando che l'avversario scelga sempre la mossa migliore per il suo gioco, ci dia il risultato migliore.

Si può capire quindi che ad ogni livello di profondità il numero di varianti cresce esponenzialmente poichè ad ogni nostra possibile mossa l'avversario ne ha altrettante per ciascuna di esse e dovremo quindi fermarci abbastanza presto dipendentemente dalle capacità dell'infrastruttura su cui eseguiamo l'algoritmo.

Notare anche che la funzione di valutazione verrà applicata soltanto all'ultimo livello di profondità per ogni ricerca perchè saranno quelli gli unici valori su cui baseremo la nostra scelta. Shannon ha categorizzato questo approccio come strategia di tipo A, nel quale si cercano tutte le varianti ad un livello di profondità fisso e non oltre.

Un miglioramento al MINMAX può essere effettuato incorporando le seguenti caratteristiche:

1) Esaminare varianti con mosse forzate più a fondo possibile e valutare solo posizioni di quasi stabilità, nei giochi in cui queste cose sono possibili.

2) Selezionare le varianti da esplorare scartandone altre che farebbero solo perdere tempo.

Shannon chiama questa strategia con le due nuove caratteristiche strategia di tipo B [2].

La complessità temporale dell'algoritmo MINMAX di tipologia A vale $O(f^p)$ quella spaziale invece vale O(fp), dove p è il livello di profondità massimo e f è il fattore di ramificazione, ovvero il numero massimo di mosse legali che possono essere effettuate in una posizione [3].

2.1 Albero di ricerca

L'albero di ricerca è una struttura astratta usata per la ricerca della soluzione di un problema. I suoi componenti sono nodi e rami; i nodi sono strutture dati che contengono, in particolare, gli stati del problema che si sta tentando di risolvere e i rami sono azioni che permettono di passare da un nodo ad un altro [3]. Nel gioco degli scacchi ad esempio, ogni stato (nodo)

corrisponde ad una diversa configurazione dei pezzi sulla scacchiera, e ogni azione (ramo) corrisponde ad una mossa che modifica lo stato e ci fa passare da un nodo ad un altro. Questo crea un collegamento tra i due nodi, un ramo collega esattamente 2 nodi, mentre un nodo può essere collegato tramite rami a più di un nodo. Un albero di ricerca dove i nodi sono stati di gioco e i rami sono mosse è anche chiamato albero di gioco.

Quando si costruisce un albero di gioco si parte sempre da un nodo chiamato radice che identifica lo stato iniziale (fig 3.1). Si aggiungono poi i vari rami che corrispondono alle mosse legali che si possono fare da quella posizione e alla fine di ogni ramo si crea un nuovo nodo che identifica la nuova posizione raggiunta mediante la mossa, questo procedimento è chiamato espansione del nodo (fig 3.2). I rami creano un rapporto gerarchico tra i due nodi che collegano: i nuovi nodi sono chiamati figli del nodo dal quale sono stati generati, che viene chiamato nodo padre. Un nodo senza figli viene chiamato nodo foglia e l'insieme di tutti i nodi foglia presenti nell'albero è definito frontiera. L'esistenza di un nodo foglia può essere giustificata da due ragioni, non esistono altre azioni che si possono fare da quella posizione o semplicemente abbiamo deciso di non espandere il nodo.

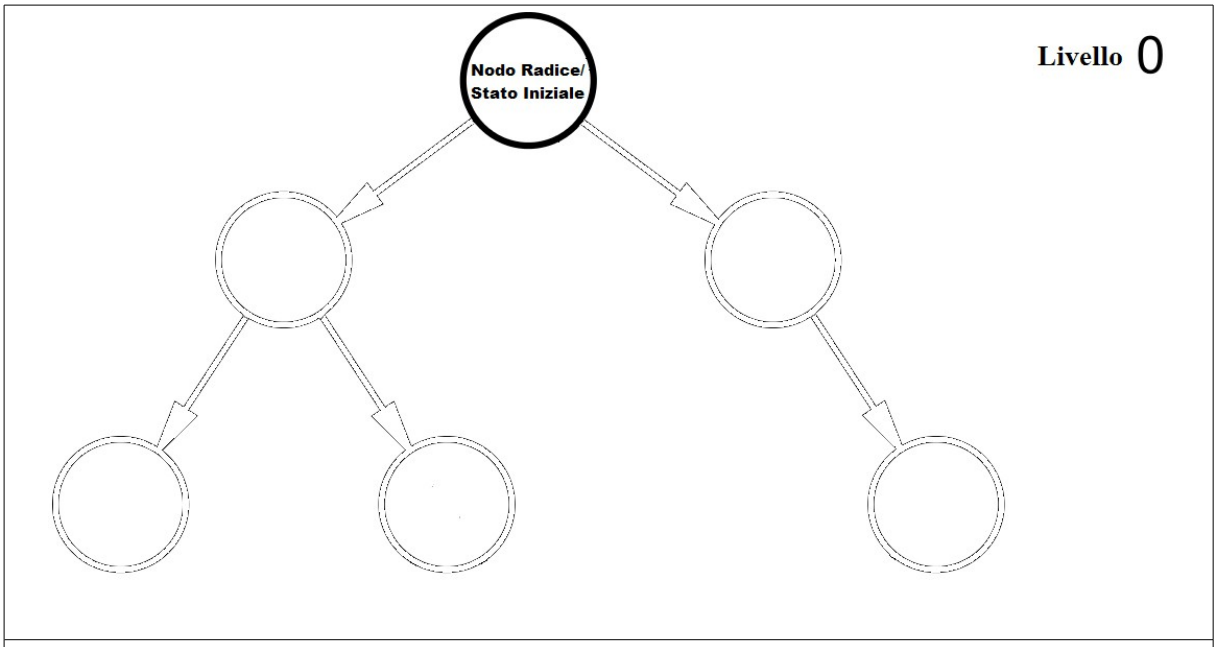

Figura 3.1 Si parte dallo stato inziale con il primo nodo, la radice. Le frecce e i cerchi vuoti sono, rispettivamente, azioni legali disponibili e stati raggiungibili tramite suddette azioni.

Fonte: elaborazione dell'autore

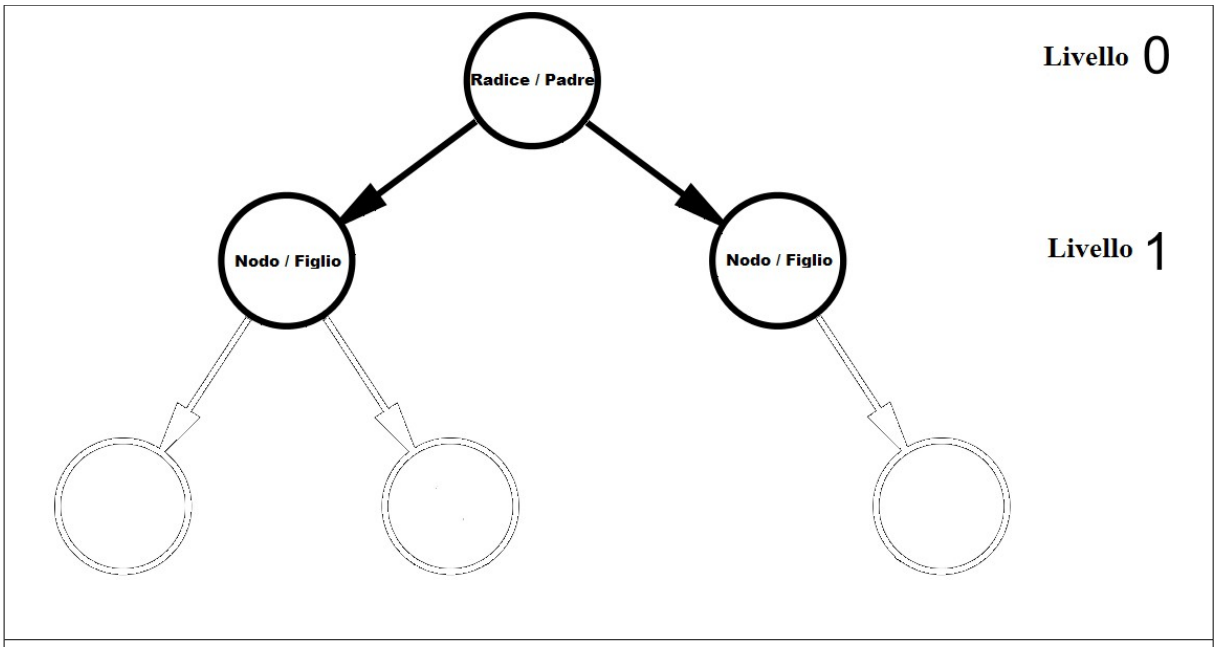

Figura 3.2 Situazione dell'albero dopo l'espansione del nodo radice e creazione nodi figlio. Le frecce piene sono i rami, ovvero le azioni che hanno prodotto i cambiamenti di stato. In questo caso i nodi al livello 1 non hanno figli, si tratta dunque di foglie che fanno da frontiera all'albero, delimitando la parte esplorata da quella non ancora esplorata.

Fonte: elaborazione dell'autore

L'insieme di tutti gli stati distinti (diversi l'uno dall'altro) che si trovano in un albero di ricerca è lo spazio degli stati, e una sequenza di nodi connessi da una sequenza di rami è un cammino. In alcune situazioni è possibile che uno stato che si trova già nell'albero di gioco possa ricomparire in seguito all'espansione di un nodo, questo può succedere in situazioni in cui le mosse sono reversibili e significa che ci sono più modi per raggiungere gli stessi stati. Secondo Russell e Norvig (p. 76) "se si è preoccupati di raggiungere il traguardo, non c'e mai nessuna ragione per tenere più di un cammino per raggiungere uno stato, perché ogni traguardo che è raggiungibile espandendo uno stato è anche raggiungibile espandendo l'altro". Possiamo quindi apportare questo miglioramento all'albero di ricerca ricordandoci quali sono gli stati già presenti nell'albero e ignorando i duplicati.

Dalla figura 3.2 possiamo capire come una struttura del genere ci possa aiutare con il nostro algoritmo MINMAX. Usando la strategia di tipo A di Shannon, se al livello 0 (quello della radice) associamo una posizione in cui tocca al giocatore A (colui che cerca di massimizzare il valore dello stato) fare la sua mossa e alle frecce le azioni legali disponibili al giocatore A, possiamo espandere il nodo radice e creare gli stati al livello 1 generati dal nodo radice. Per ogni stato del livello 1, dove tocca al giocatore B (colui che cerca di minimizzare il valore dello stato) fare la sua mossa, facciamo la stessa cosa, creiamo gli stati raggiungibili da ogni stato del livello 1, tramite azioni legali, espandendone i nodi e creando il livello 2. Finito con il livello 2 tocca di nuovo al giocatore A ed espandiamo ora tuti i nodi del livello 2 creando il livello 3. Andiamo avanti cosi con i turni che si susseguono finché non si raggiunge il livello di profondità prefissato.

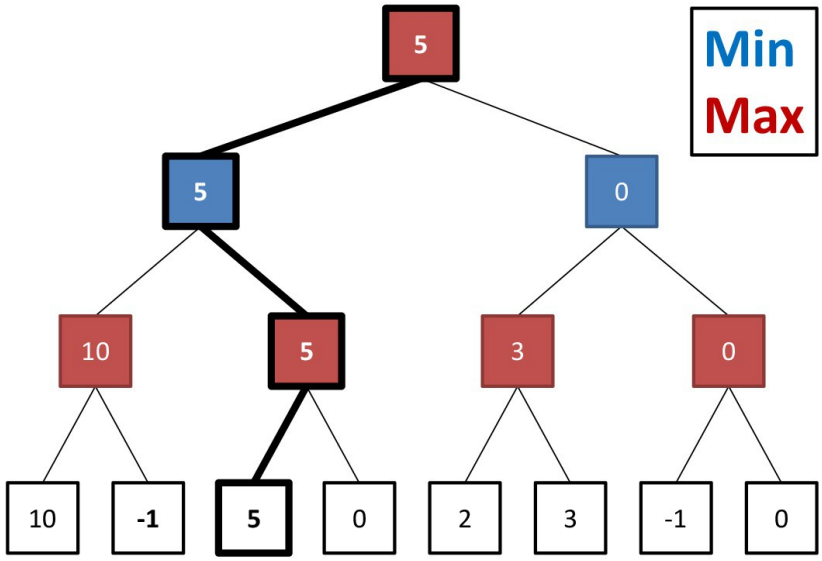

Figura 3.3 Illustrazione dell'algoritmo MINMAX tramite albero di gioco con valori di stato delle foglie arbitrari e in seguito propagati tramite suddetto algoritmo fino alla radice.

Fonte: Yannakakis & Togelius (2018, p. 44)

Nella figura 3.3, dove gli stati di colore rosso (turni pari, livello 0 e livello 2) sono le posizioni in cui tocca al giocatore A fare la propria mossa e quelli blu (turni dispari, livello 1 e livello 3) gli stati dove tocca al giocatore B, possiamo vedere questa espansione fino al livello 3 dove si trovano le foglie di colore bianco. Abbiamo dato ad ogni stato della frontiera un valore arbitrario e ne abbiamo propagato i valori ai livelli superiori seguendo la logica del MINMAX descritta al capitolo 3. La sequenza di stati e rami in grassetto è la variante principale (percorso migliore disponibile) scelta in base al valore che ha raggiunto la radice sfruttando l'algoritmo MINMAX.

2.2 Valutazione della posizione

Una valutazione quantitativa dello stato è fondamentale per il funzionamento dell'algoritmo MINMAX, valutare una posizione statica in un gioco significa assegnare a quello stato, tramite una funzione euristica, quando una funzione di valutazione che dia il valore esatto non

è disponibile, un valore numerico che è un'approssimazione della soluzione che si cerca. Quindi se si vuole sapere se una posizione è vinta, persa, o un pareggio si possono per esempio assegnare a questi stati rispettivamente i valori 1, -1, e 0 (a volte si assegnano anche $+\infty$ alla vittoria e $-\infty$ alla sconfitta), se una funzione di valutazione esatta fosse disponibile restituirebbe uno di quei tre valori. In assenza di una funzione del genere la funzione euristica restituisce un valore n per cui $-1 < n < 1$.

Nel gioco degli scacchi una funzione che riesca a dirci se la posizione è vinta, persa, o un pareggio ai primi stadi della partita al momento non esiste per via dell'elevata complessità a cui abbiamo accennato nel paragrafo 2.6 [2].

Esistono tuttavia giochi per cui questo è possibile e tali giochi si definiscono risolti [1].

I giocatori di scacchi durante le partite sono in grado di dare una valutazione qualitativa ad una posizione usando alcune regole non scritte che non sono altro che consigli pratici per giocare bene, nate dall'esperienza accumulata dai giocatori nel corso della storia del gioco. Abbiamo per esempio il controllo del centro della scacchiera, il valore relativo dei pezzi in base alla loro forza/mobilità , la struttura generale dei pezzi sulla scacchiera o la loro mancanza perché catturati dall'avversario ecc..

Poichè non siamo in grado di dare una valutazione perfetta dobbiamo tentare di emulare il modo in cui un giocatore umano valuta una posizione, o usare altre euristiche, sfruttando le sopracitate massime o altre informazioni valide e pertinenti che potrebbero aiutare.

Shannon propone un esempio di funzione valutazione per una posizione *P* nel gioco degli scacchi: "f(*P*) = 200(K-K') + 9(Q-Q') + 5(R-R') + 3(B-B'+ N-N') + (P - P') - 0.5 (D-D' + S - S' + I-I') + 0.1(M-M') + ... ;

(1)K,Q,R,B,B,P sono i numeri di re, regina, alfieri, cavalli, pedoni bianchi sulla scacchiera

(2)D,S,I sono i pedoni bianchi doppiati, arretrati, isolati.

(3)M = mobilità del bianco (misurata come il numero di mosse legali disponibili al bianco); le lettere accentate sono le equivalenti per il nero. I coefficenti 0.5 e 0.1 sono stime dello scrittore." (Shannon p. 6) La funzione restituisce un valore che andremo ad assegnare alla posizione valutata che useremo in seguito per decidere la variante principale tramite l'algoritmo MINMAX.

Un modo per migliorare il processo di valutazione è tramite l'uso di *hash table*, che sono tabelle che contengono posizioni di gioco già valutate in precedenza [10]. Grazie a queste tabelle possiamo spendere meno tempo nella valutazione e andare più a fondo nella ricerca.

Una buona funzione euristica ci permette di prendere decisioni migliori e ridurre il tempo di ricerca indicandoci in alcuni casi e con l'uso di altri algoritmi in sinergia con il MINMAX, i percorsi su cui investire più tempo e risorse nell'esplorazione, evitandoci di eplorare posizioni che ci farebbero solo perdere tempo.

2.3 Esplorazione Albero

Abbiamo presentato nel paragrafo 3.1 l'albero di ricerca, il suo funzionamento e come poi viene sfruttato dall'algoritmo MINMAX. L'esplorazione dell'albero per la ricerca della soluzione, avviene tramite l'espansione dei nodi sulla frontiera, all'inizio ne abbiamo solo uno, il nodo radice che rappresenta lo stato di partenza, che una volta espanso arricchisce l'albero con molti altri nodi in relazione alle azioni che si possono compiere dallo stato iniziale. Ora dobbiamo continuare la ricerca espandendo i nodi appena creati, ma sorge una perplessità, quale nodo espandere per primo? Nel caso della radice non avevamo questo problema poiché avendo un solo nodo non avevamo scelta, ma ora ci troviamo davanti a una moltitudine di possibili percorsi alcuni più vicini e altri più lontani dalla soluzione cercata.

"E' largamente sostenuto che la maggiorparte, se non tutta, l'IA sia in realtà solo ricerca. Quasi ogni problema di IA può essere visto come un problema di ricerca, che può essere risolto trovando il miglior (in accordo con alcune misure) piano, percorso, modello, funzione, ecc. Gli algoritmi di ricerca sono pertanto spesso considerati come il cuore dell'IA." (Yannakakis & Togelius p. 39)

L'argomento dei prossimi 3 sottoparagrafi sono proprio gli algoritmi di ricerca il cui compito è quello di costruire l'albero seguendo un ordine preciso nella scelta dei nodi da espandere per primi (che è ciò che differenzia i vari algoritmi) partendo dalla radice.

Gli algoritmi di ricerca sono divisi da un altro aspetto, alcuni non hanno informazioni addizionali sulla soluzione ma sono in grado di distingure uno stato soluzione da uno che non lo è, e si chiamano algoritmi di ricerca non informati, questi algoritmi non avendo informazioni addizionali sul traguardo non sono visti come IA. Gli altri posseggono qualche informazione aggiuntiva sulla soluzione del problema e sono chiamati algoritmi di ricerca informati. Gli algoritmi esposti a seguire sono *deph-first* (prima-in-profondità) e *breadh-first* (prima-in-ampiezza) che sono algoritmi non informati e gli algoritmi *best-first* (prima-il-migliore) che sono informati [5].

2.3.1 Depth-first

Esattamente come suggerisce il nome (prima-in-profondità) questo algoritmo effettua la costruzione dell'albero espandendo prima in profondità. Significa che espande per primi i figli del nodo che ha appena espanso finché non raggiunge un nodo terminale (foglia che non può più essere espansa perché non ci sono azioni legali possibili) oppure un nodo soluzione, a quel punto tenta di espandere i nodi della stessa generazione (generati dallo stesso nodo padre) e se riesce continua ad andare in profondità altrimenti se ha provato tutti i nodi di quel livello e sono tutti terminali ma nessuno è la soluzione riprende l'espansione dal livello superiore dimenticando ogni volta i nodi terminali che sono stati scartati. Siccome non tiene in memoria i percorsi creati da ricerche fallimentari significa che un algoritmo *deph-first* tiene in memoria soltanto il percorso corrente e i nodi generati sui vari livelli del percorso corrente ma non ancora analizzati. Si può pensare a questo algoritmo come ad una pila in cui l'ultimo elemento ad entrare, ovvero l' ultimo nodo ad essere generato, è il primo ad uscire, ovvero viene espanso per primo.

Un problema che questo algoritmo però può incontrare, è quello di ritrovarsi in un ciclo continuo e questo accade quando ci sono stati ripetuti che possono essere raggiunti l'un l'altro. Per ovviare a questo problema ci sono 2 soluzioni; la prima è quella di evitare i nodi duplicati sul percorso corrente e la seconda è quella di limitare la ricerca ad una profondità fissa, in figura 3.4 possiamo vedere l'espansione di un albero binario (numero massimo di figli per nodo è limitato a 2) con profondità 3. Quest'ultima soluzione però, crea un'altro problema: l'effetto orizzonte, che vedremo più avanti.

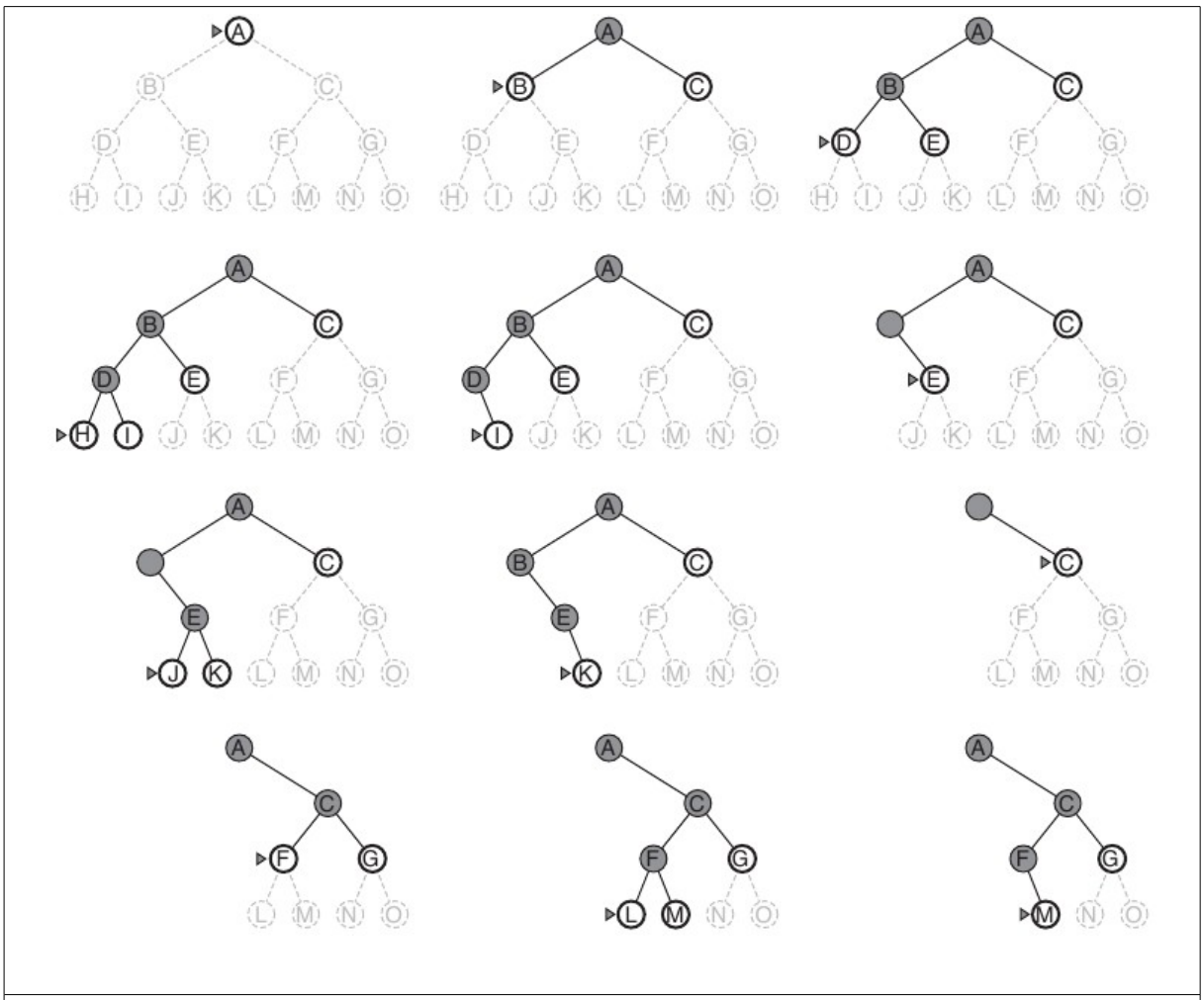

Figura 3.4 Albero binario con profondità 3, in grigio scuro il percorso corrente, il triangolino/freccia indica il nodo che sta venendo valutato per l'espansione, i nodi bianchi in chiaro sono i nodi espansi e non ancora valutati, quelli tratteggiati sono possibili percorsi futuri di nodi non ancora espansi. I nodi e rami scomparsi sono percorsi fallimentari e quindi eliminati. Su ferma su M perché si è deciso che M è soluzione in questa illustrazione.

Fonte: Russell & Norvig (2009, p. 86)

Un algoritmo di ricerca *deph-first* in un albero in cui esista un limite di profondità ha di conseguenza una complessità dello spazio pari a O(pf). La complessità temporale invece, vale $O(f^p)$.

Nel nostro caso però ci interessa usare questa strategia di espansione insieme all'algoritmo MINMAX per cercare la variante migliore che ci dica la mossa da fare.

Quando si usa *depth-first* per i giochi non-cooperativi a due giocatori intanto bisogna per forza dare un limite di profondità altrimenti dovremmo esplorare l'intero spazio degli stati e anche di più se teniamo stati duplicati, e abbiamo già chiarito nel capitolo 3 che questa non è una strada praticabile. Poi, invece di fermarci quando si trova un nodo soluzione (vittoria) dobbiamo comunque valutare tutti i nodi foglia sulla frontiera perché a meno che non sia una serie di mosse forzate prima di arrivare in quella posizione toccherà al nostro avversario e non ci permetterà di raggiungere tale stato. Quando si raggiunge un nodo foglia si valuta con la funzione di valutazione e poi si fa lo stesso con gli altri nodi generati dallo stesso padre, poi se il padre è uno stato in cui la mossa tocca al giocatore che vuole minimizzare allora il valore del padre sarà il minimo tra quelli dei figli altrimenti sarà il massimo. I nodi figli si dimenticano ogni volta che si usa il loro valore per decidere il valore dei padri. Si continua facendo lo stesso per gli altri nodi foglia e i loro padri dopodichè si confrontano man mano i valori dei padri che propageranno il minimo o il massimo ai livelli superiori, si continua cosi finchè non si sono valutati tutti i nodi foglia [3].

2.3.2 Breadth-first

L'algoritmo *breadth-first* cerca la soluzione ad un problema espandendo tutti nodi di un livello prima di passare al successivo. Il primo è sempre il nodo radice che dopo l'espansione crea il livello 1 con tutti i nodi raggiungibili dallo stato iniziale. A questo punto l'algoritmo espande tutti i nodi del livello 1 generando i loro figli e creando il livello 2. Va avanti cosi generando un livello dopo l'altro finché non trova un nodo soluzione, figura 3.5. Questo algoritmo assicura di trovare la soluzione più veloce, ovvero più vicina al nodo radice, che si traduce nel percorso meno costoso.

L'algoritmo funziona come una coda, i primi nodi ad entrare sono anche i primi ad uscire, significa che i figli dei nodi appena espansi finiscono alla fine della coda dei nodi che devono essere espansi [3].

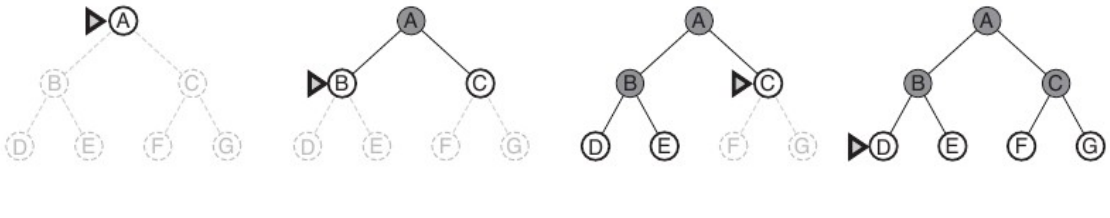

Figura 3.5 Albero binario con profondità 2, in grigio scuro i nodi espansi. Il triangolino indica il prossimo nodo che verrà espanso. I nodi bianchi sono i figli dei nodi espansi fin'ora e rappresentano la frontiera.

Fonte: Russell & Norvig (2009, p. 82)

A differenza del depth-first, breadth-first memorizza tutti i nodi che ha esplorato e questo è il problema maggiore di questo algoritmo che ha una complessità dello spazio $O(f^p)$ e temporale $O(f^p)$ [11]. Anche in questo caso, più facciamo andare in profondità l'algoritmo, migliore è la valutazione di MINMAX, ma abbiamo gia visto diverse volte che bisogna mettere un limite di profondità all'algoritmo. Una volta che sono stati espansi tutti i nodi fino alla profondità scelta -1 perche i nodi sulla frontiera (che è l'ultimo livello) non devono essere espansi ma solo valutati, MINMAX ha a disposizione i nodi da dare in pasto alla funzione di valutazione e iniziare a propagare i valori su fino alla radice.Possiamo notare che finchè i nodi da valutare non si rendono disponibili non possiamo applicare MINMAX quindi dobbiamo aspettare che l'algoritmo crei l'intero albero fino alla profondità che abbiamo scelto, solo a quel punto possiamo cominciare. Altrimenti potremmo applicare il MINMAX ad ogni livello che espandiamo ma la valutazione fattà fino a quel momento diventerebbe superflua poiché la valutazione migliore è sempre quella del livello più profondo, quindi tanto vale applicare la funzione di valutazione solo a quei nodi. L'algoritmo MINMAX funziona per sua natura in modalità depth-first, anche le loro complessità temporali e spaziali sono uguali, quindi usare un'espansione *breadth-first* in questo caso non è ottimale e ad essa è preferibile *depth-first* anche per via del risparmio di memoria [3].

2.3.3 Best-First

Gli algoritmi *best-first* hanno informazioni ulteriori oltre a quelle specificate dal problema e le usano per decidere quali nodi espandere per primi, si parla di informazioni di natura esterna al problema come le informazioni che si usano per la valutazione euristica della posizione, è propria quest'ultima che gli algoritmi *best-first* sfruttano per prendere le loro decisioni. Alcuni algoritmi *best-first (*come A*), oltre alla funzione di valutazione euristica usano anche altre funzioni per decidere il valore del nodo da espandere.

L'idea base è quella di individuare lo stato migliore, chiamato anche foglia principale (che è la foglia che determina il valore della radice), tra quelli disponibili ed espanderlo. Ogni volta che si espande una foglia principale si ricalcola il valore MINMAX confrontando tutti i nodi sulla frontiera, il nodo il cui valore raggiunge la radice diventa la nuova foglia principale ed è il prossimo nodo ad essere espanso.

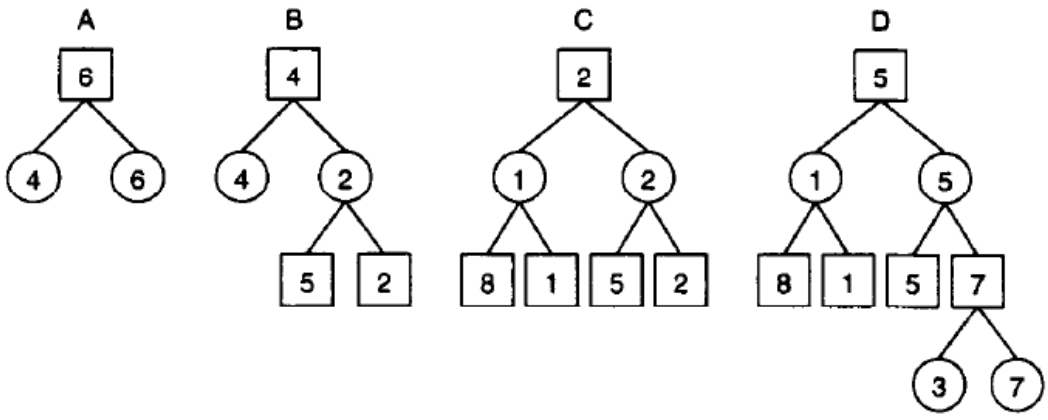

Figura 3.6 Albero binario con ricerca *best-first*. Nodi quadrati sono nodi di massimo e i cerchi sono nodi nodi di minimo.

Fonte: Korf & Chickering (1996, p. 301)

Come si può vedere in figura 3.6, nella situazione A dove è stato espanso solo il nodo radice il valore migliore è il 6 quindi è quella la nostra foglia principale per adesso e viene quindi espansa e lo possiamo vedere nella situazione B in cui i nodi generati hanno valore 5 e 2. A questo punto si calcola il valore MINMAX della radice tra 4, 5 e 2 che in questo caso è il 4 e quello stato diventa la foglia principale che verrà espansa in seguito come si può vedere nella situazione C. In C la foglia principale diventa il 2 in basso a destra che porta poi alla sua espansione e all'albero raffigurato nella situazione D. L'algoritmo mantiene in memoria tutti i nodi man mano che li genera perché ogni volta che espande un nodo deve cercare il nodo foglia principale successivo tra tutti i nodi della frontiera [12]. La complessità temporale e spaziale di best-first è al massimo $O(f^p)$, valore che può esser ridotto in base all'accuratezza della funzione di valutazione [3].

2.4 Tagli Alfa-Beta

Uno dei problemi principali di MINMAX in cui ci siamo imbattuti fin'ora è la limitazione della profondità di ricerca causata dalla crescita esponenziale del numero di nodi in relazione al livello di profondità. Per mitigare questo problema sarebbe utile ridurre il numero di nodi che devono essere presi in considerazione, in modo da ottenere un albero di gioco più snello permettendoci di analizzare i nodi nello stesso spazio temporale di quanto si farebbe normalmente con una strategia di tipo A di Shannon, ma ad un livello di profondità maggiore, oppure allo stesso livello di profondità ma in tempo minore. Ogni nodo che si trova nell'albero di gioco è li perché rappresenta una posizione dalla quale se ne raggiungono molte altre che potrebbero avere valori interessanti per il nostro gioco, quindi per decidere quali rami potare dal nostro albero per renderlo più snello, senza rischiare di eliminare percorsi significativi,

possiamo affidarci all'algoritmo Alfa-Beta [13].

In generale se ci trovassimo con un nodo in qualunque parte dell'albero che abbia un valore X per un giocatore A che deve fare la mossa per arrivare a tale nodo con valore X, ma esiste un nodo, raggiunbile dal giocatore A, in un livello superiore o con lo stesso padre del nodo con valore X, che abbia un valore migliore di X dal punto di vista del giocatore A, allora il nodo con il valore X non verrà mai raggiunto, fig 3.7 [3].

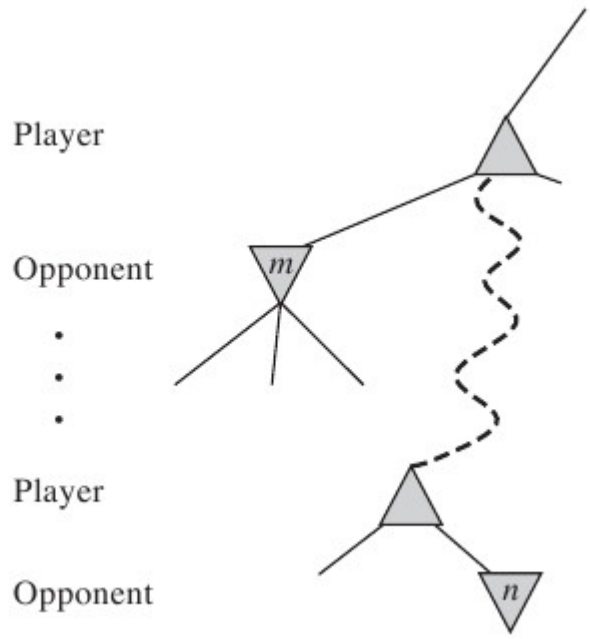

Fig 3.7 Se *m* è un nodo con un valore migliore di *n* per Player (giocatore le cui mosse stiamo analizzando) *n* non verrà mai raggiunto.

Fonte: Russell & Norvig (2009, p. 169)

In dettaglio questo algoritmo utilizza due variabili chiamate Alfa e Beta il cui compito è quello di fare da intervallo dei valori ammissibili con estremi non compresi (significa che un valore che sia uguale ad uno degli estremi non essendo migliore verrà scartato). Alfa è il limite inferiore e all'inizio viene settato al valore più basso possibile (generalmente $-\infty$) e Beta è il limite superiore e viene settato al valore più alto possibile (generalmente $+\infty$).

Il giocatore che cerca di minimizzare i valori delle posizioni diminuirà il valore di Beta, invece il giocatore che cerca di massimizzare il valore delle posizioni aumenterà il valore di Alfa, in ogni momento quindi Alfa sarà il valore migliore trovato fin'ora per il giocatore massimizzante e Beta per quello minimizzante. Questi estremi vengono aggiornati su tutto il percorso corrente fino alla radice, significa che associamo questi estremi ad ogni nodo che sia un nodo padre e quando si valuta un figlio il suo valore va ad aggiornare l'intervallo Alfa-Beta del padre modificando Alfa se quest'ultimo è un nodo di massimo o Beta se è un nodo di minimo. Quando gli estremi di un nodo di cui abbiamo cominciato a valutarne i figli delimitano un intervallo che si trovi fuori dall'intervallo di un nodo di livello superiore

(l'intersezione dei due intervalli è formata al massimo da 1 elemento che può essere uno dei due estremi dell'intervallo dei nodi di livello superiore) possiamo smettere di valutare i figli rimanenti perché significa che un valore uguale o migliore è gia stato scoperto, altrimenti si valutano tutti i figli e si aggiornano le variabili Alfa e Beta del padre che una volta valutati tutti i nodi figlio diventeranno uguali, non essendoci altri nodi figlio che modifichino l'intervallo, e potremo collassarli in uno solo e si continua a propagare sui nodi superiori in accordo con il valore del nodo di cui abbiamo valutato i figli, modificando Alfa se siamo in un nodo di massimo o Beta se siamo in un nodo di minimo.

Nel momento in cui non ci saranno più foglie da valutare i valori Alfa e Beta della radice si troveranno ad essere uguali a quelli del figlio migliore e troveremo la variante principale che sarà il percorso dalla radice alla foglia che per prima ha assegnato il valore corrente alla radice [13].

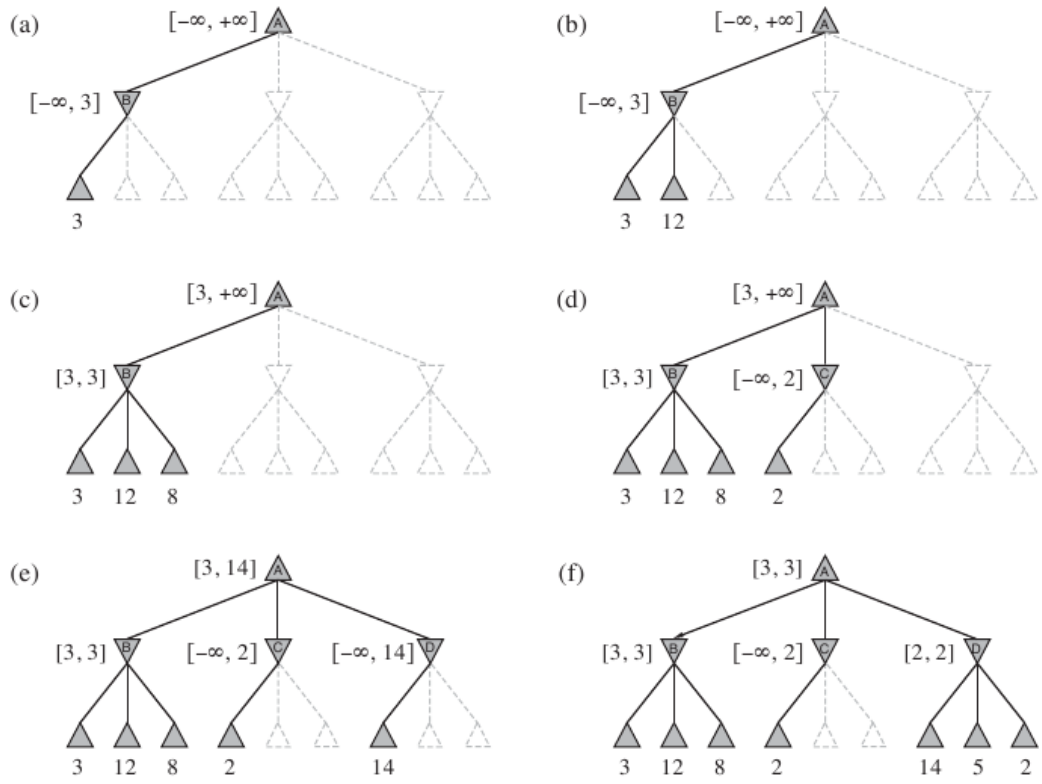

Fig 3.8 Applicazione di Alfa-Beta a profondità 2. Le foglie tratteggiate nello stato (f) sono state potate dall'algoritmo.

Fonte: Russell & Norvig (2009, p. 168)

Prendiamo in considerazione la figura 3.8 che vediamo sopra, possiamo notare come il valore delle due foglie potate non sia rilevante al fine di determinare il valore della radice tramite MINMAX perché essendo C un nodo di minimo e avendo scoperto che uno dei suoi figli vale 2, significa che il nodo C non potrà mai valere più di 2. L'intervallo dei valori di C nella situazione (d) diventa quindi (-∞, 2) che si trova fuori dall'intervallo di valori delimitato dal

suo nodo padre, che in questo caso è il nodo radice A il cui intervallo vale (3, +∞) . Di conseguenza avendo già trovato un valore migliore di 2 da assegnare al padre di C, possiamo evitare di valutare gli altri nodi di C [3].

2.4.1 Move Ordering

L'algoritmo Alfa-Beta dà il meglio di se quando il primo dei nodi figlio di un nodo padre appena espanso è anche la posizione migliore tra quelle disponibili: valore maggiore se padre è massimizzante e valore minore se padre è minimizzante.

In figura 3.8 ad esempio, dal nodo padre D si raggiungevano 3 posizioni dal valore 14, 8 e 2, il nodo D è un nodo di minimo quindi il valore migliore tra quelli disponibili è quello del terzo figlio. Se il nodo di valore 2, che è il migliore in questo caso, si fosse trovato ad essere valutato per primo invece che per terzo, le mosse possibili in D fossero dunque state ordinate dalla migliore alla peggiore, l'intervallo del nodo padre D sarebbe uscito dall'intervallo di valori ammissibili dei suoi nodi superiori immediatamente. In quel caso, dato che il valore 2 sarebbe stato il massimo possibile per il nodo D e non rientrando nell'intervallo di valori delimitato da Alfa-Beta, non avremmo dovuto continuare con la valutazione degli altri 2 nodi foglia e ci saremmo trovati dunque nella stessa situazione del nodo C risparmiandoci tempo prezioso. Precisamente se riuscissimo a valutare per primi sempre i nodi migliori, Alfa-Beta ci farebbe esaminare solo $O(f^{p/2})$ nodi invece dei $O(f^{p})$ di MINMAX.

Se fosse possibile ordinare le mosse dalla migliore alla peggiore senza informazioni sulle posizioni da valutare la funzione di ordinamento potrebbe essere usata per giocare una partita perfetta, quindi possiamo capire che un ordinamento del genere non è possibile senza informazioni aggiuntive. Alcuni metodi per ordinare le mosse includono usare informazioni raccolte in ricerche precedenti per trovare la mossa migliore, che viene anche definita mossa killer per via del fatto che "uccide" le altre mosse dato che non verranno esaminate, oppure, negli scacchi per esempio, provare per prime le mosse che danno scacco o che catturano un pezzo avversario con uno di minor valore o mosse che minacciano di dare scacco o catturare un pezzo di valore nei turni a seguire ecc, o una qualunque funzione euristica che sia adatta alla situazione in cui ci si trova [3].

2.4.2 Iterative deepening

L'*Iterative deepening* (approfondimento iterativo) è un metodo di esplorazione degli alberi di ricerca che unisce *depth-first* e *breadth-first,* il che gli permette di trovare la soluzione più vicina alla radice e quindi il percorso più corto grazie alle caratteristiche di *breadth-first* e di avere la complessità spaziale di *depth-first*. L'esplorazione dell'albero viene eseguita in modalità *depth-first* con il livello di profondità della ricerca variabile che si incrementa di 1 ogni volta che si sono valutati tutti i nodi alla profondità corrente e non si è trovata una soluzione. Quest'ultima caratteristica, il fatto che esamini tutti i nodi di un livello prima di passare al successivo, è ciò che lo rende *breadth-first*. All'inizio il limite di profondità è inizializzato a 0, parte quindi controllando se lo stato iniziale sia una soluzione o meno, in

caso lo sia conclude la ricerca altrimenti aumenta il limite di profondità di 1 ed espande il nodo iniziale creando il primo livello e ne controlla i nodi. Se neanche li trova una soluzione ricomincia dall'inizio incrementando il livello di profondità massimo a 2 dopodiché ricomincia ad eseguire una tipica esplorazione in stile *depth-first* come abbiamo visto nel sottoparagrafo 3.3.2. Se neanche al livello 2 trova niente incrementa di nuovo il limite di profondità e ricomincia la sua ricerca dal nodo radice sempre in modalità *depth-first.* Questa ricerca iterativa continua ad aumentare il livello di profondità finché non trova una soluzione [11]. Possiamo vedere una ricerca fino al livello di profondità 3 in figura 3.9.

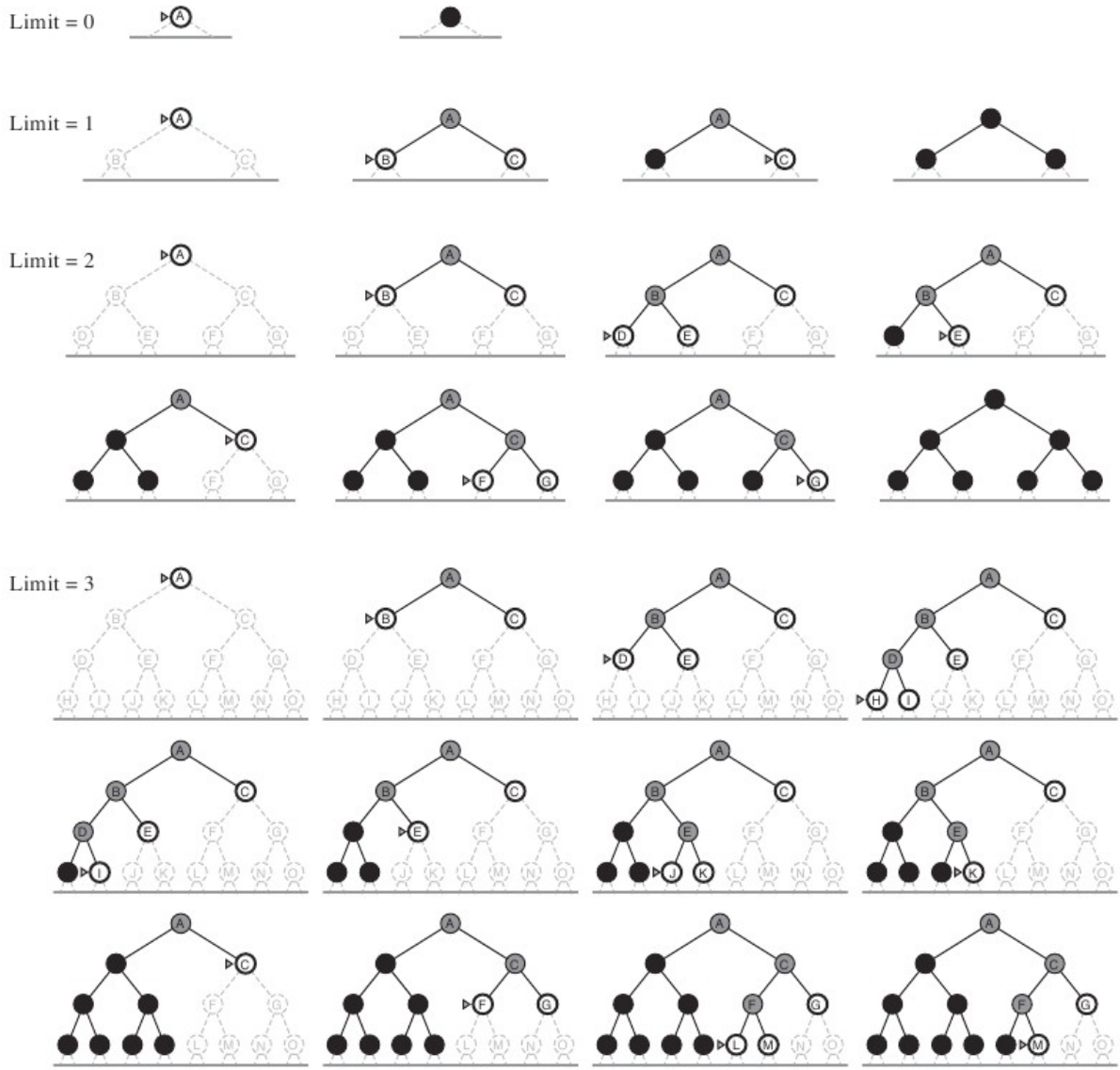

Fig 3.9 *Iterative Deepening* su un albero binario fino alla quarta iterazione (livello di profondità 3) con soluzione trovata al nodo M.

Fonte: Russell & Norvig (2009, p. 89)

Verrebbe da pensare che valutare continuamente gli stessi stati sia uno spreco di risorse e di tempo, ma siccome la maggiorparte dei nodi si trova sulla frontiera e la loro crescita è esponenziale, il numero di nodi ai livelli superiori non è paragonabile al numero dei nodi sulla frontiera e siccome questi ultimi sono generati una sola volta e quelli al livello immediatamente superiore 2 volte e quello sopra ancora 3 volte e cosi via fino alla radice, il numero di nodi in più da valutare non è cosi alto perché avvicinandosi alla radice, dove si trovano i livelli che vengono valutati più volte, il numero dei nodi da valutare diminuisce in modo esponenziale.

Uno dei vantaggi del fatto che *Iterative deepening* valuti più volte le stesse mosse, si ha quando viene utilizzato insieme ad Alfa-Beta, e la ragione sta nel fatto che ad ogni iterazione siamo a conoscenza dei valori dei nodi che andremo ad analizzare in seguito, e questo ci permette di ordinare le mosse in modo da valutare per prime le mosse killer, riducendo il numero dei nodi da valutare ad ogni iterazione in modo significativo, avvicinandoci alla complessità temporale ideale per Alfa-Beta dal valore di $O(f^{P/2})$, che senza un buon ordinamento non è facile da raggiungere [3].

2.4.3 Transposition Table

In alcuni giochi ci sono stati che possono essere raggiunti usando una sequenza di mosse diverse fra loro o in cui cambia semplicemente l'ordine di esecuzione di queste, ci si riferisce a questi percorsi diversi con cui si raggiunge lo stesso stato con il nome di trasposizioni. Abbiamo accennato nel paragrafo 3.2 alle *hash tables* come un modo per ottimizzare la valutazione degli stati sfruttando le valutazioni effettuate in passato, le proprietà principali di queste tabelle sono la velocità elevata con cui si eseguono operazioni su di esse e l'assegnazione casuale di indirizzi che puntano ad elementi salvati in esse che può anche portare all'assegnazione dello stesso indirizzo per due elementi diversi. Le *transposition tables* (tabelle di trasposizione) sono strutture dati, normalmente implementate come *hash tables* per velocizzare l'accesso ai contenuti, che memorizzano le posizioni codificate che fungono da indirizzo e i relativi valori aggiornati calcolati tramite funzione euristica e ancora, il livello di profondità nel quale sono state valutate. Un'altra cosa che possiamo salvare per ogni stato è la mossa migliore disponibile in quella posizione che è stata trovata in una ricerca passata. Gli stati sono aggiunti alle tabelle, se si decide di memorizzarli, dopo essere stati valutati. Se si tenta di inserire una stato che ha generato un indirizzo hash già presente nella tabella ci si trova di fronte ad una collisione che possiamo risolvere decidendo se mantenere il valore precedente o aggiornarlo, di solito si aggiorna se la valutazione corrente avviene ad un livello pù profondo. La consultazione invece avviene prima dell'espansione di un nodo per vedere se informazioni su tale posizione sono state memorizzate in precedenza e si agisce in accordo a ciò che si trova.

Con un database del genere possiamo migliorare di molto la profondità a cui si può cercare con MINMAX e ancora di più se si usano le *transposition tables* per l'ordinamento delle mosse necessario all'ottimizzazione di Alfa-Beta [10].

Abbiamo già visto però che lo spazio degli stati di giochi complessi è estremante elevato quindi non possiamo memorizzare tutte le posizioni anche se questo ci porterebbe immensi benefici, dobbiamo quindi decidere quali stati tenere e quali rimuovere usando la strategia più

appropriata [3].

2.5 Effetto orizzonte

Uno degli effetti negativi che si riscontrano durante le ricerche con limite di profondità è chiamato effetto orizzonte. Si tratta di un comportamento particolare, simile al procrastinare degli esseri umani. Quando l'algoritmo si trova in una situazione dove è presente un problema che non riesce a risolvere, per esempio negli scacchi una mossa dell'avversario inevitabile e molto negativa per noi contro cui non abbiamo una risposta adeguata, decide di fare altre mosse che magari peggiorano la situazione attuale, sacrificando eventualmente pezzi minori, pur di non affrontare il problema. Continua a posporre la perdita inevitabile finché questa non si trova oltre al limite di ricerca prefissato uscendo di fatto dalle situazioni possibili e analizzabili dall'algoritmo e dando l'illusione che il problema non esista più, quando in realtà è solo stato spostato oltre l'orizzonte degli eventi che l'algoritmo è in grado di vedere e molto probabilmente se lo ritroverà di nuovo alla ricerca successiva [3].

Non c'e modo di eliminare completamente questo difetto causato dal limite fissato in cui fermare la ricerca, dato che, come abbiamo già visto più volte, non possiamo fare a meno di mettere questo vincolo di profondità. Qualcosa che si può fare però c'è, possiamo per esempio usare le estensioni singole, che consitono nell'espansione ulteriore di mosse a cui è stata riconosciuta la qualità di "essere "indubbiamente migliori" rispetto ad altre in una data posizione. Una volta scoperta in qualunque parte dell'albero durante la ricerca questa singola mossa viene memorizzata. Quando la ricerca raggiunge il limite normale, l'algoritmo controlla per vedere se la singola estensione è una mossa legale; se lo è, l'algoritmo ne permette la considerazione. Queste rende l'albero più profondo, ma siccome ci saranno poche singole estensioni, non vengono aggiunti molti nodi totali all'albero" ([3] p.174).

2.6 Ricerca quiescente

La ricerca quiescente ha come scopo quello di trovare uno stato di quiete in cui nulla di catastrofico o estremamente positivo possa succedere nel futuro prossimo. Quello che si vuole evitare, è di ritrovarsi in una posizione che magari sembra buona in rispetto alla valutazione data dall'euristica quando in realtà si è ad un passo dal dare, per esempio, uno scacco matto all'avverario o di riceverlo, perdendo la partita. Per decidere se una posizione è quiescente o meno non si usa la stessa funzione che si usa per dare al nodo la valutazione necessaria all'algoritmo MINMAX, altrimenti non sarebbe necessario fare un'ulteriore analisi, si usano invece altre funzioni euristiche che guardano a diverse caratteristiche rispetto a quelle sfruttate dall'euristica di valutazione esposta nel paragrafo 3.2. Questo tipo di ricerca dello stato di quiete avviene espandendo i nodi della frontiera che vengono categorizzati come non quiescenti. Si tratta quindi di andare un po più a fondo rispetto al limite fissato per la ricerca, almeno 2 mosse in più ma non più di 10 [2]; dobbiamo quindi concentrarci solo su proprietà

cruciali limitando il numero di nodi da espandere ulteriormente, per risolvere velocemente il problema.

Bisogna anche notare, che applicare la funzione di valutazione a posizioni che non sono quiescenti, è uno spreco dato che verranno ulteriormente espanse, senza contare che probabilmente riceverebbero un valore errato, quindi prima bisogna vedere se uno stato è di quiete o meno, perché in quella situazione anche la valutazione sarebbe più corretta . La ricerca quiescente è anche un valido aiuto al problema dell'effetto orizzonte, contribuendone alla mitigazione [3].

2.7 Varianti

Introducendo alcune modifiche alla struttura di MINMAX possiamo adattarlo per altre situazioni, o migliorarne alcuni aspetti. Ad esempio fin'ora la valutazione dei nodi sulla frontiera è stata fatta esclusivamente dal punto di vista del giocatore massimizzante, ma c'e anche la possibilità di valutare i nodi dal punto di vista del giocatore che deve fare la mossa. Se invece ci trovassimo di fronte ad un gioco con elementi stocastici dovremmo integrare in qualche modo questa nuova caratteristica in MINMAX, dato che quello che abbiamo visto fin'ora copre soltanto i giochi deterministici.

2.7.1 Negamax

Quando si tratta di analizzare giochi a più giocatori, cambiare continuamente punto di vista può complicarne la comprensione, per questo MINMAX viene esposto normalmente dal punto di vista del giocatore che cerca di massimizzare il proprio utile. Ma quando si tratta di esporlo usando formalismi matematici o implementarlo scrivendo un programma software, è più elegante e comodo usare Negamax.

Questa piccola modifica, consiste nel valutare i nodi sempre dal punto di vista del giocatore che deve eseguire un'azione, e al momento di propagare i valori fino alla radice, cambiare il segno (negazione) delle valutazioni ad ogni livello, e scegliere sempre il nodo con il valore più alto. Quindi se p è una posizione da valutare, allora $f(p)$ è la valutazione della posizione dal punto di vista del giocatore che si ritrova a dover eseguire un'azione partendo dalla posizione p.

Dopo la valutazione dei nodi sulla frontiera, bisogna decidere quale valore dei figli verrà assegnato ai nodi padre. In MINMAX se un nodo padre è un nodo minimizzante si deve scegliere la foglia con l'utile minore, in Negamax invece, non ci sono nodi minimizzanti, sono tutti massimizzanti! Quindi in ogni momento si tratta di scegliere il valore maggiore tra quelli disponibili. Per poter essere in grado di scegliere sempre il valore maggiore, prima si cambiano i segni dei valori assegnati ai nodi figlio, e poi tra quei valori con segno cambiato, si assegna al padre il valore massimo. Si continua cosi, cambiando segno ai nodi figlio prima

di decidere il valore massimo da dare ai padri, fino ad assegnare un valore alla radice, a quel punto, avremo la nostra variante principale e la prima mossa da fare.

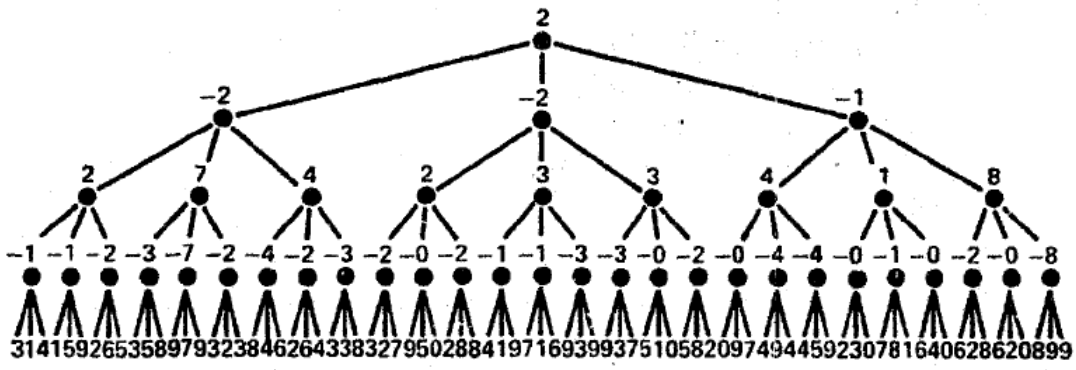

Fig. 3.10 Albero di gioco a profondità 4 con valori Negamax.

Fonte: Knuth & Moore (1975, p. 299)

In figura 3.10 possiamo vedere l'algoritmo Negamax applicato ad un albero di profondità 4. Prendiamo i primi 3 nodi foglia sulla sinistra dal valore di 3, 1 e 4. Sappiamo che quei valori sono stati ottenuti tramite la funzione di valutazione applicata dal punto di vista del giocatore che deve muovere da quella posizione, quindi sappiamo che il padre di quei nodi è una posizione in cui muove l'altro giocatore, essendo i turni alternati, di conseguenza quei valori vengono cambiati di segno per invertirne l'ordine di preferenza e poi scegliere quello con il valore più alto. In questo caso diventano -3, -1, -4, e tra questi il valore maggiore è -1 ed è proprio quello che viene assegnato al padre di quei nodi foglia essendo il migliore disponibile. Poi continuiamo con il padre del nodo a cui abbiamo appena dato il valore -1, e dobbiamo decidere tra i suoi figli che hanno utili equivalenti a -1, -1, e -2, ne cambiamo i segni ed essi diventano 1, 1 e 2, e il maggiore fra questi, cioè 2, viene assegnato al padre di turno. Si continua cosi fino alla radice.

Per quanto riguarda l'implementazione di Alfa-Beta con Negamax, la differenza con MINMAX si trova nel fatto che bisogna alternare le posizioni di Alfa e Beta ad ogni livello successivo e bisogna anche cambiarli di segno (quindi per ogni livello, Alfa diventa -Beta e Beta diventa -Alfa) [13].

2.7.2 Expectiminmax

Quando si tratta di giochi con elementi stocastici il MINMAX visto fin'ora non è adeguato perché non prevede dei meccanismi per gestire l'influenza della casualità sulle conseguenze delle azioni. La presenza di elementi aleatori richiede quindi l'aggiunta di ulteriori componenti all'albero, oltre a quelli soliti necessari per operare con MINMAX; in questo caso si tratta di nodi di probabilità. Come si può vedere in figura 3.11, da ogni nodo di probabilità

(nodi a forma di cerchio) escono dei rami che rappresentano uno degli eventi distinti possibili, generati dall'elemento aleatorio, e ad essi sarà assegnato un valore pari alla probabilità di verificarsi di tale evento, per esempio, se il nostro elemento aleatorio è un dado a sei facce, gli eventi disponibili sono le varie facce del dado che hanno tutte probabilità pari a 1/6, quindi ci saranno sei rami uscenti dal nodo di probabilità atti a rappresentare questi 6 eventi con le loro rispettive probabilità (notare che la somma delle probabilità dei rami uscenti da un nodo di probabilità deve sempre essere uguale a 1). Alla fine di ognuno dei suddetti rami ci saranno i nodi di minimo e di massimo rispettivamente, in cui il giocatore di turno dovrebbe decidere la propria azione tra quelle legali disponibili in conseguenza a quale evento aleatorio si è verificato. Poi dai nodi di minimo e massimo escono i rami in cui all'altro estremo ci sono i nodi di probabilità, uno per ogni azione legale disponibile in quel nodo di massimo o minimo.

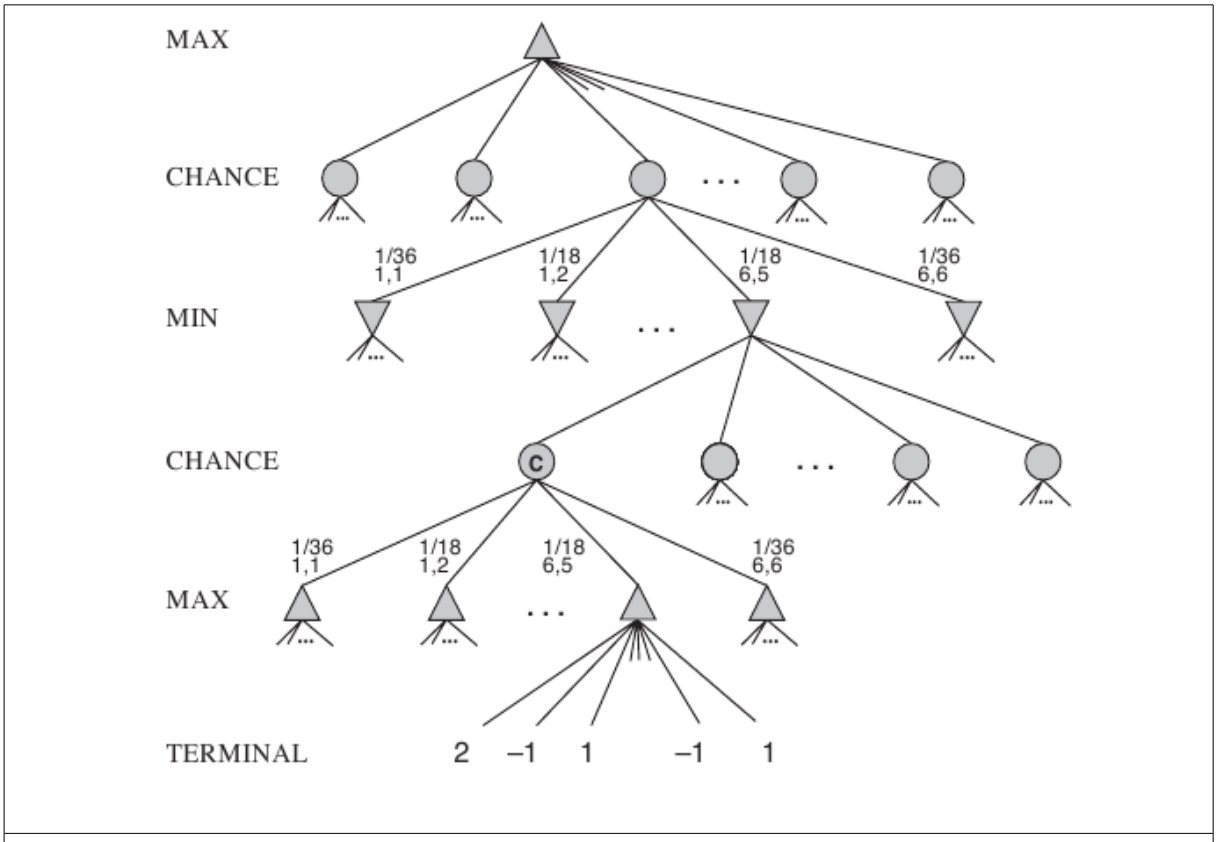

Fig. 3.11 Albero di ricerca Expectimax, con due dadi a sei facce come elemento aleatorio.

Fonte: Russell & Norvig (2009, p. 178)

Ora, per calcolare il valore da propagare alla radice, i nodi di minimo, massimo, e le foglie, si comportano come nel MINMAX, quindi le foglie vengono valutate con una funzione euristica e i nodi massimizzanti e minimizzanti prendono il minimo o il massimo dai loro nodi figlio (che in questo caso sono sempre nodi di probabilità).

Ai nodi di probabilità invece, viene assegnato un valore, chiamato valore previsto, che è la somma dei prodotti dei valori dei figli per la probabilità del rispettivo ramo : $\Sigma_r \; p(r) \; f(n_r)$ (dove r sono i rami, $p(r)$ è la probabilità assegnata ad ogni ramo, $f(n_r)$ è il valore dei nodi figlio). La complessità temporale di questo algoritmo è peggiore di quella di MINMAX a

causa dei nodi di probabilità, e vale $O(f^p n^p)$, dove n è il numero di eventi distinti resi disponibili dall'elemento aleatorio [3]. E' possibile migliorare la ricerca dell'algoritmo Expectiminmax usando Alfa-Beta per i nodi di minimo e massimo; i nodi di probabilità invece, possiamo potarli integrando gli algoritmi sviluppati da Ballard, Star1 e Star2 [14].

3 Algoritmo MCTS

La limitatezza della profondità di ricerca e la difficoltà nella costruzione di una buona funzione di valutazione euristica sono alcuni dei difetti principali di MINMAX.

MCTS (*Monte-Carlo Tree Search*) è un algoritmo di ricerca *Best-First* iterativo, in grado di vincere i sopracitati problemi e allo stesso tempo lavorare bene anche in ambienti non-deterministici e a informazione imperfetta, e con abbastanza potere computazionale, MCTS è in grado di approssimare l'albero di gioco di MINMAX.

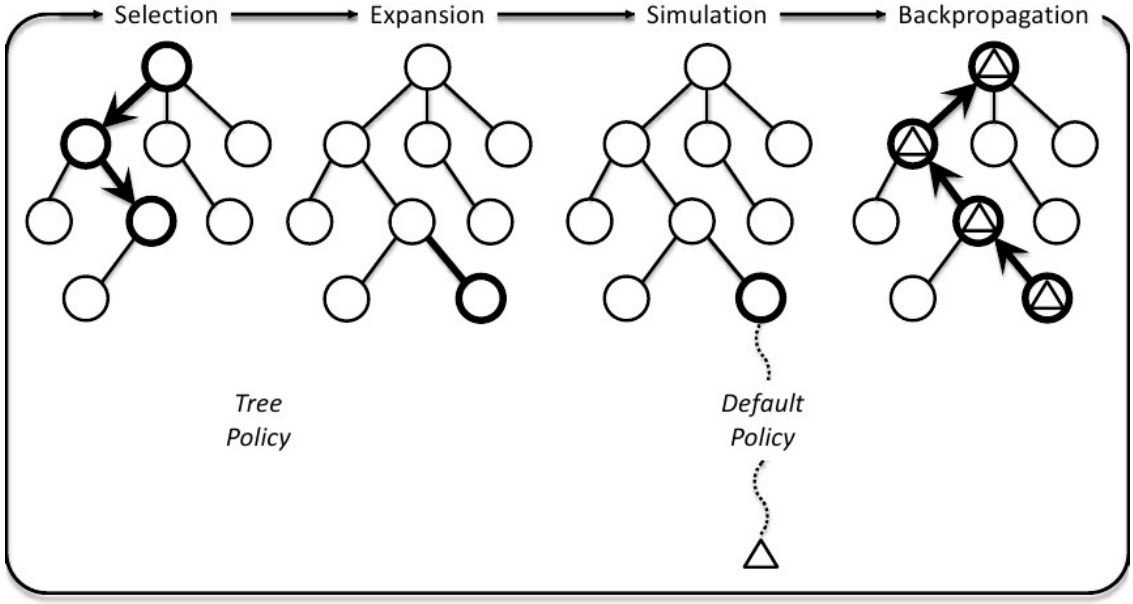

Fig 4.1 Le quattro fasi di MCTS: Selezione, Espansione, Simulazione, Propagazione.

Fonte: Yannakakis & Togelius (2018, p. 47)

A rendere MCTS *Best-First* è la scelta dei nodi da espandere durante la prima fase dell'algoritmo, che è la **selezione** e che parte sempre dal nodo radice. Questa avviene sfruttando vari metodi tra cui: *UCB1*, *espilon-greedy*, *Thompson sampling* o *Bayesian bandits*. Usando una di quelle formule si seleziona il nodo che possiede il potenziale maggiore e che abbia almeno 1 figlio non espanso.

Dopo la selezione del nodo più promettente, si passa alla seconda fase, l'**espansione**, che consiste nell'aggiungere all'albero di ricerca corrente e quindi salvare in memoria, uno dei

figli del nodo selezionato nella prima fase, scelta che spesso avviene in modo casuale tra i figli disponibili.

Dall'aggiunta del nuovo nodo all'albero, si passa alla terza fase di MCTS, la **simulazione**. Partendo dal nodo figlio designato, si scelgono azioni legali, di solito in modo casuale (ci sono varianti di MCTS dove si usano altri approcci in questa fase), finchè non si raggiunge uno stato finale, ovvero dove non ci sono più mosse legali disponibili caratterizzato da una vittoria, sconfitta o pareggio a cui viene assegnato un valore dato da una funzione di valutazione che sia in grado di riconoscere e valutare uno stato terminale, eseguendo di fatto, una simulazione di partita attraverso mosse casuali. I nodi incontrati durante questo percorso simulato non vengono aggiunti all'albero.

Una volta ottenuta la valutazione del nodo terminale, la si propaga sul percorso che è stato seguito nella fase 3 fino alla radice. A differenza di MINMAX però, dove i valori sostituiscono quelli precedenti, in MCTS i valori propagati si sommano a quelli correnti, assegnando ai nodi dell'albero dei valori che sono una stima di tutte le simulazioni fatte, in cui quei nodi sono partecipi; questi sono poi i valori che vengono sfruttati nella prima fase per la selezione del nodo migliore. La quarta fase appena vista, quella della **propagazione**, e le precedenti 3 sono illustrate in figura 4.

Alla fine della quarta fase l'algoritmo ricomincia dalla selezione per un nuovo ciclo, comportamento che rende MCTS iterativo e gli conferisce un'altro vantaggio rispetto a MINMAX, la possibilità di essere fermato in qualunque momento [5].

4 Conclusioni

Abbiamo visto che l'algoritmo MINMAX è in grado, teoricamente, di arrivare ad una soluzione terminale se gli viene dato abbastanza tempo e in alcuni giochi questo è possibile. Per giochi più complessi però, anche con un alto potere computazionale si necessita di tempi troppo lunghi per permettere all'algoritmo di esplorare tutto lo spazio degli stati, si parla addirittura di 10^{90} anni per trovare la prima mossa nel gioco degli scacchi se si calcola una variazione per microsecondo [2]. Ma nonostante questo, grazie alle tecniche di miglioramento come Alfa-Beta, una buona funzione euristica, *Iterative Deepening* e tanti altri, si possono comunque ottenere ottimi risultati.

Uno dei primi usi di Alfa-Beta risale al 1962 quando uno studente di John McCarthy implementò l'algoritmo in un programma scacchistico che ha poi preso parte al primo match disputato tra due computer e aveva come avversario ITEP, un programma scritto all'istituto di fisica teorica e sperimentale di Mosca che faceva altresì uso di Alfa-Beta, scoperto in modo indipendente da Alexander Brudno [15]. L'incontro è avvenuto tramite telegrafo ed è finito con la vittoria per 3 a 1 di ITEP. In seguito ci sono stati altri programmi scacchistici che hanno vinto diversi premi, ma il più famoso è stato il Deep Blue di IBM che usava proprio Alfa-Beta con una ricerca a profondità prefissata che è riuscito a battere il campione del mondo Garry Kasparov nel 1997.

Attualmente uno dei motori scacchistici più forti è Stockfish, un programma *open source* che

fa uso di potatura Alfa-Beta e di tutte le ottimizzazioni descritte in precedenza in questa tesi e molte altre. Nel 2017 Stockfish ha preso parte ad una sfida composta di 100 partite contro AlphaZero, la creazione del DeepMind team di Google che fa uso di *Deep Learning* e MCTS, la sfida è finita con la vittoria di AlphaZero [16].

Tutt'oggi però, rimane ancora insoluto il problema dell'effetto orizzonte anche se le tecniche disponibili riescono a mitigarne un po gli effetti. In alternativa si possono usare algoritmi come MCTS che si comportano in modo più simile agli umani con la loro ricerca *Best-First,* anche se nel caso degli scacchi per esempio, MINMAX e Alfa-Beta si comportano meglio rispetto a MCTS (se non si considera l'uso insieme a *Deep Learning* come nel caso di AlphaZero) a differenza di giochi come GO in cui MCTS è migliore. Altri algoritmi a cui possiamo dare attenzione sono A*, B*, SSS* oppure ulteriori miglioramenti di Alfa-Beta come Scout o NegaScout se si usa Negamax.

Bibliografia

[1] Allis, L. V. (1994). Searching for Solutions in Games and Artificial Intelligence. PhD thesis, Univ. Limburg, Maastricht, The Netherlands.

[2] Shannon, C. E. (1950). XXII. Programming a computer for playing chess. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, *41*(314), 256-275.

[3] Russell S., & Norvig P. (2009), *Artificial Intelligence: A Modern Approach*, Prentice Hall Press, Upper Saddle River, NJ.

[4] Morgenstern, O., & Von Neumann, J. (1953). *Theory of games and economic behavior.* Princeton university press.

[5] Yannakakis, G.N., & Togelius, J. (2018). *Artificial Intelligence and Games*. Springer, Heidelberg.

[6] Peleg, B., & Sudhölter, P. (2007). *Introduction to the theory of cooperative games* (Vol. 34). Springer Science & Business Media.

[7] Nash, J. (1951). Non-cooperative games. *Annals of mathematics*, 286-295.

[8] Chalkiadakis, G., Elkind, E., & Wooldridge, M. (2011). Computational aspects of cooperative game theory. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, *5*(6), 1-168.

[9] Brocas, I., Carrillo, J. D., & Sachdeva, A. (2018). The Path to Equilibrium in Sequential and Simultaneous Games: a Mousetracking Study.

[10] Zobrist, A. L. (1970). Technical Report #88. Computer Science Department, University

of Wisconsin, Madison, WI.

[11] Korf, R. E. (1985). Depth-First Iterative-Deepening: An Optimal Admissible Tree Search. *Artif. Intell., 27*, 97-109.

[12] Korf, R. E., & Chickering, D. M. (1996). Best-first minimax search. *Artificial intelligence*, *84*(1-2), 299-337.

[13] Knuth, D. E., & Moore, R. W. (1975). An analysis of alpha-beta pruning. *Artificial intelligence*, *6*(4), 293-326.

[14] Ballard, B. W. (1983). The*-minimax search procedure for trees containing chance nodes. *Artificial Intelligence*, *21*(3), 327-350.

[15] Alexander Brudno (1963). *Bounds and valuations for shortening the search of estimates*. Problemy Kibernetiki (10), 141–150.

[16] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., ... & Lillicrap, T. (2017). Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*.